The DevOps Adoption Playbook: A Guide to Adopting DevOpsin a Multi-Speed IT Enterprise
By Sanjeev Sharma

CHAPTER 1

DevOps: An Overview

RANT OF A DEVELOPMENT MANAGER

So, the developer completes writing code for a new service by Monday afternoon. She builds the code, runs unit tests, and delivers the code to the integration stream so it gets included in the continuous integration (CI) build. To get her service tested, before leaving for work, she opens a ticket with the Quality Assurance (QA) team.

Tuesday morning, the QA team comes in and sees the ticket assigned to them. A tester gets the ticket and emails the developer asking for the deployment instructions. As there is no deployment automation, the developer responds saying she will deploy the service to the QA environment herself. Tuesday afternoon, they get on a conference call to deploy the code. The developer discovers that test environment is not compatible with her code. They need a new environment. Tuesday evening, the tester opens a ticket with the operations (Ops) team for a new environment, with the new specs.

Wednesday morning, the Ops team assigns the ticket to an engineer who looks at the specs and sees a firewall port change. As he leaves for lunch, he opens a ticket with the security team to approve the port change. Wednesday afternoon, the security team assigns the ticket to a security engineer, who approves the change. Wednesday evening, the Ops engineer receives the approval and starts building the new environment. He needs to manually build new Virtual Machines (VMs), with an Operating System (OS), app server, database, and web server.

Thursday morning, the server build is done, and the ticket is closed. The tester emails the developer again to deploy the new service. The developer deploys the service, and the tester starts walking through the test scripts, which pass. He now needs to run a regression test but needs additional test data to re-run tests. Thursday afternoon he opens a ticket to request new test data with the production support team.

continued

continued

> Friday morning, the production support team assigns a database analyst (DBA) to extract test data from production. But now it's Friday afternoon. Everyone knows DBAs don't work on Friday afternoons. Monday morning, the tester gets the test data from the DBA. It takes him 20 minutes to run the regression tests and discover a defect. He returns the ticket to the developer—a full week after the code was written and built. A full week of coding has now been done on top of that code, not knowing it was defective. We are now another week behind.
>
> What's scary about this story is that when I tell it to my peers in other companies, they shake their heads not in empathy but in amazement as to how efficient we are compared to them!
>
> —Yet another frustrated development manager

DevOps: Origins

The DevOps movement began with a seminal talk given by John Allspaw and Paul Hammond (both at Flickr/Yahoo at that time) at the O'Reilly Velocity 2009 conference. The talk was entitled "10+ Deploys Per Day: Dev and Ops Cooperation at Flickr."[1] Ten deploys a day was considered unprecedented. Their approach was eventually referred to as *DevOps* by Patrick Debois, when he organized the first *DevOpsDays* event in Ghent, Belgium, the same year.

While the name caught on and started getting tremendous interest, the traction was initially limited to startups, more specifically, organizations delivering web applications. These applications were created by developers (the Dev) who typically delivered changes and updates to their web apps in a very rapid manner. The main hurdle they faced was that of operations (the Ops), which were slow in deploying those changes, as they had rigid and rigorous change management processes.

The goal of the DevOps movement was to address this impedance mismatch between the Dev and Ops teams; to bridge the chasm between them; and to foster more communication, collaboration, and trust. At its heart, it was a cultural movement, focused on changing the cultural differences between Dev and Ops, along with automation to make application delivery faster, more efficient,

[1] http://conferences.oreilly.com/velocity/velocity2009/public/schedule/detail/7641

and eventually, continuous. In 2010, Jez Humble, then at ThoughtWorks, took DevOps to practitioners throughout the industry with his book *Continuous Delivery*, codifying some of the practices that make up the core of DevOps and making DevOps adoption tangible and available to all.

Still, DevOps was seen as something done by the *unicorns*—the startups and the upstarts, organizations at the cutting edge of innovation, without large, complex legacy systems to maintain. It had not yet gone mainstream with the large enterprises. However, these large enterprises were seeing what the startups were achieving with DevOps, and were trying to determine how to adapt DevOps for their own needs. Organizations like IBM were beginning to dabble with deployment automation, and with visual architecting of environments, and even stitching these two capabilities together. At the same time, well-established companies in the build automation space, like UrbanCode, started pivoting into DevOps with the release of uDeploy, thus establishing a new category of tools to enable continuous delivery. Other companies in the automation space, like Nolio, joined in with their own competitive offerings. In parallel, coming from the Ops and *infrastructure as code* side, companies like Opscode (now called Chef) and Puppet Labs were gaining traction (Opscode with Chef, and Puppet Labs with Puppet).

The real growth for DevOps into large enterprises began in 2012, with companies like IBM jumping into the fray with their first, albeit short-lived, continuous delivery experiment with SmartCloud Continuous Delivery. Several consulting firms, like ThoughtWorks and IBM, also started to offer consulting services for organizations, especially large enterprises looking to adopt DevOps, and helping to translate what worked for the unicorns so that it could work for enterprises. IBM and CA Technologies announced their formal entrance into the DevOps world by acquiring UrbanCode and Nolio, respectively (and coincidently on the same day in April 2013). However, the biggest turning point for the DevOps movement since its inception came later, in 2013, with the publication of Gene Kim's book, *The Phoenix Project*. This book, inspired by and modeled after the historic *The Goal* by Eliyahu M. Goldratt, became the must-read book for the modern-day implementation of *Lean* practices and Goldratt's *Theory of Constraints* in the IT world, just as Goldratt's book had been a few decades earlier for the manufacturing world. Kim truly took DevOps mainstream with his book, as well as subsequent work he has done with the *State of DevOps Report* that he publishes every year, with Jez Humble and Puppet Labs.

DevOps: Roots

Where does DevOps come from? While I have already outlined its origin story, the true roots of DevOps predate Allspaw, Debois, Humble, and Kim by almost a century. You have to go way back to the 1910s and look at the origins of the Lean movement.

The Lean movement started in manufacturing with Henry Ford and his adoption of Lean for flow management in the Model T production lines. This work was further extended, refined, and codified by Kiichiro Toyoda and Taiichi Ohno at Toyota starting in the 1930s and really accelerating after World War II. Their work was both refined and influenced by Dr. William E. Deming in the 1950s, who proposed the Plan–Do–Check–Act (or Adjust) cycle (PDCA), to continuously improve manufacturing quality. Based on this core approach, the Lean manufacturing movement aimed to both continuously improve the product being manufactured and reduce waste in the manufacturing process. Lean was further refined in the works of James P. Womack and Daniel T. Jones when they published *The Machine that Changed the World* in 1990 and (required reading for everyone) *Lean Thinking* in 1996 (Lean.org, 2016).

DEMING ON LEAN THINKING AND CONTINUOUS IMPROVEMENT

> Dr. W. Edwards Deming taught that by adopting appropriate principles of management, organizations can increase quality and simultaneously reduce costs (by reducing waste, rework, staff attrition and litigation while increasing customer loyalty). The key is to practice continual improvement and think of manufacturing as a system, not as bits and pieces.
>
> —Dr. Deming's Management Training (Deming, 1998)

In 2001 came Agile, a group of 17 thought leaders, including Alistair Cockburn and Martin Fowler, who created *The Agile Manifesto*.[2] The core principles of the manifesto were to get away from the rigid, waterfall-oriented, documentation-heavy world of software development, which was resulting in most software development projects being late, over budget, or abject failures.

[2] http://www.agilemanifesto.org

Their goal was to move to an iterative approach where there was constant interaction with the customer, end-user, or a surrogate who represented them. They wanted to move away from measuring progress through major rigid milestones such as *Requirements Documentation*, which brought code no closer to being delivered than the day before. Other goals were to use real running code (working software) as the true measure of progress; to look at planning as being adaptive to real progress; and to create requirements that did not need to be written in stone up front, but would evolve and be refined as the applications were being developed.

Agile was refined with the development of methodologies like XP, Scrum, and, more recently, Scaled Agile Framework or SAFe. Today, Agile is used by both large and small organizations to deliver projects of all sizes and technologies.

Agile was the precursor to, and became the core driver for, the need for DevOps. As developers started delivering code faster, that code needed to be tested faster; it also needed to be deployed to Dev and test servers, and eventually to production, more often. The Ops teams were not set up for this, which resulted in a major bottleneck being created at the Dev-to-test handoff, due to lack of availability of the right test environments as and when needed and, more importantly, at the hands of production at release time. Production release remained a major undertaking, with "release weekends" that typically lasted beyond the weekend.

THE RELEASE WEEKEND

I remember when I was working as a developer at a financial services institution in the early '90s. (We called them banks back then.) On release weekends, much to my chagrin, we were asked to show up at work on Friday mornings with our sleeping bags in hand. We were expected to stay there through the weekend. There were multiple conference rooms set up with conference call bridges open to get every team in communication with each other. One conference room was set up like a war room with the project leader coordinating all the stakeholders off a massive spreadsheet. The management did their best to create a party atmosphere, but that faded right after the first few hours. We were communicating with the Ops people for the first time. We were handing off our code to people who had never

continued

continued

seen the code before. They were deploying code into environments we had no visibility into, using scripts and tools we had no familiarity with. It would be chaos the whole weekend. Lots of delivered food and stale coffee, and nothing seemed to work as planned. And the traders we supported, they were smart. They always planned their team outing or picnics on the Monday following. They knew nothing would work. And they were right. Fortunately, we only did this twice a year. Even more fortunately for me personally, I worked there for only two releases.

The rapid development of code in short iterations amplified the need for better collaboration and coordination between Dev and Ops teams. The frequent failure of release to production exposed the need for providing developers with access to production-like environments. The major inefficiencies in the entire process were exposed by making just one part of the process—developing code—more efficient, which created major bottlenecks with test and Ops. If you think of the application development and delivery process as an assembly line in a factory, speeding up an operation of just one of the stations to increase the number of widgets it produces does not help the overall delivery speed if the downstream stations are still operating at a slower speed. It just creates more of a backlog for them. (See Figure 1-1.) This was not just a challenge for Ops, but for all the stakeholders in the delivery life cycle.

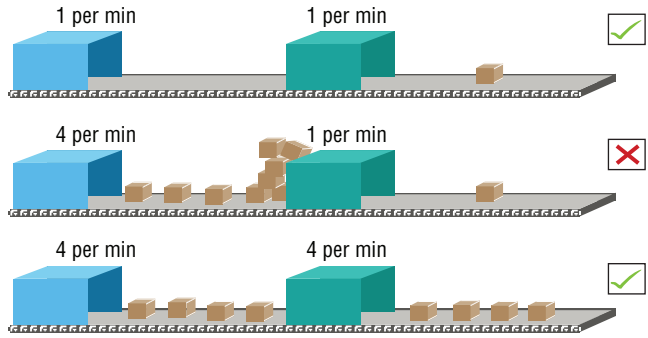

Figure 1-1: Delivery pipeline bottlenecks

The focus now turned toward minimizing *cycle time*—the time from the inception of a requirement, or *user story*, to the time that capability is in the hands of the customer, or at least is integrated, tested, and ready to be deployed to the customer. This resulted in the development of the two core

capabilities of DevOps: continuous integration (already a core competency of Agile) and continuous delivery. I will discuss both capabilities in detail shortly. This extension of Agile beyond the Dev-test cycle—including the Ops team in the delivery cycle, as a part of the process, and not in a separate silo that was not engaged until the code was ready for release—became the core principle of DevOps.

Addressing Dev versus Ops

Dev and Ops have traditionally lived in different silos, with misaligned, even opposing priorities. Development (Dev) is tasked with creating innovation and getting it into the hands of users as soon as possible. Operations (Ops) is tasked with making sure that the users have access to a stable, fast, and responsive system. While Dev and Ops' ultimate goal is to make the user a satisfied (and potentially a happy, paying) customer of the systems they provide, their views of how to do it tend to be inherently antithetical. No developer wants to intentionally produce a buggy system that would cause the application to crash while a user is using it. No operations person wants developers to not produce updates with new, exciting features and capabilities. It is how they go about it that is different. This is a classic symptom of what is referred to as *water-Scrum-fall* (Forrester, 2011). Developers want, and are expected to produce, new features quickly. Operations want, and are expected to produce, a stable system, at all times.

Dev versus Ops

Before the advent of Agile, in the purely waterfall-oriented paradigm, when developers and operations lived in truly isolated worlds, these opposing priorities were not that much of an issue. Developers and operations worked on a schedule that was marked by limited interactions, only at release times. Developers knew when the release date was, and they could only release new features then. If they did not create a new feature by the release date, they would have to wait for the next *release train*. Operations knew when the train would come to town. They would have enough time to test the new features before deploying them, and they could take days (weekends) to deploy them out to customers. For large systems, they could even deploy in a phased manner spread over long periods of time. Stability was maintained.

Agile changed all that. With continuous integration (CI), developers were now deploying their features daily. There was no release train to wait for; it

was a conveyer belt (pipeline) that ran all the time. The developers now wanted their features up and running—in the Dev environment, in the test environment, and finally in production (Prod)—at the same frequency at which they produced and integrated them. They wanted Ops to accommodate all these new releases.

Ops now had to deal with not one release every so often but a continuous barrage of CI builds. These builds may or may not have been deployment-ready, but they had to be managed by Ops and deployed to test, and eventually production, environments. Ops now cared more about quality. Developers and testers cared about how quickly they could get Dev and test environments and whether or not those environments were production-like. They did not want to test the code they were delivering on environments that did not function and behave like production environments. Thus, Ops could no longer take days to provision and configure new environments—for Dev, test, and eventually Prod. They had to do all of this while still maintaining stability and reliability of production systems.

CYCLE TIME?

If you have two-week Scrums but it takes three weeks to get a new test server, how long are your Scrums?

Dev and Ops

The solution to this battle between Dev and Ops is what DevOps addresses: to achieve the balance between innovation and stability and between speed of delivery and quality. To achieve this, both Dev and Ops need to improve how they operate and align.

The Dev View The previous section may give the impression that Ops needs to change more than Dev, but Dev also needs to make several changes:

- Dev needs to work with Ops to understand the nature of the production systems their applications will be running on. What are the standards for the production systems (environment patterns) and how should their applications perform on them? Within what constraints do the applications need to operate? Dev now needs to understand system and enterprise architectures.

- Dev needs to get more involved in testing. This means not just making sure that their code is bug-free but also testing the application to see how it will perform in production. This requires Dev to work closely with Quality Assurance (QA) and to test their application in a production-like system. (I'll discuss production-like systems later in this chapter.)
- Dev also needs to learn how to monitor deployed applications and understand the metrics Ops cares about. They need to able to decipher how processes interact and how one process can cause another one to slow down or even crash. They need to understand how changes to their code will impact the entire production system and not just their own application.
- Dev needs to communicate and collaborate better with Ops.

The Ops View Ops needs to be able to provision new environments rapidly, and they need to architect their systems to absorb rapid change.

- Ops needs to know what code is coming and how it may impact their system. This requires them to be involved with Dev, right from understanding requirements and system specs of the applications being developed. This process is referred to in Lean and DevOps as *shift left*. They need to make sure that their systems can accommodate these applications as they are enhanced.
- Ops needs to automate how they manage their systems. Rapid change with stability cannot be achieved without automation. Automation will allow not only rapid change but also rapid rollbacks, if something does break.
- Ideally, Ops needs to version their systems. This can only be done when the infrastructure and all changes to it are captured and managed as version-controlled code. Thus, they need to leverage infrastructure as code or, even better, software-defined environments. (I'll talk more about that later in this chapter.)
- Ops needs to monitor everything throughout the delivery pipeline, whichever environment the Ops teams manage. They need to be able to spot potential instability as soon as it happens.
- Ops needs to communicate and collaborate better with Dev.

In a nutshell, Dev and Ops both need to be brought into the DevOps paradigms. They both need to know that this is not going to be easy, or something

that can be achieved in a day. They need to plan for and work toward gradually adopting the changes needed to achieve the promise of DevOps. They may never get to—and in most cases should never get to—where Dev and Ops are one team, but they need to understand that their roles will change as they adopt DevOps. They need to change enough to work together and find the right alignment between Dev and Ops that their organization needs and improve from there.

That being said, the gap between Dev and Ops is not the only inhibitor to a fast cycle time in the delivery lifecycle. All the stakeholders in the delivery lifecycle need to communicate and collaborate better.

The Business View Let's look at the *business view*. At the end of the day, it is the business's requirements that IT is delivering through the applications and services delivered. What does the business (lines of business to be more precise) need?

- Business needs visibility into the status of what is being delivered by IT. Are they on time and on budget to deliver the applications and services?
- Business needs the application delivery teams to provide feedback on how the clients and end users are utilizing the applications and services being delivered. Are they able to get the business value as expected by the business?

A more detailed analysis on the business's point of view and expectations of IT, and how DevOps helps the business will be discussed in detail in subsequent chapters.

DevOps: Practices

Much has been written in books, and even more in blog posts, about the capabilities that make up DevOps. Several thought leaders have divided these practices into various categories, and in some cases even with different names. IBM lists several such practices, which are found under the following broad categories:

- Think
- Code
- Deliver
- Run
- Manage

- Learn
- Culture

This taxonomy comes from the IBM Garage Method,[3] a new methodology for adopting DevOps focused on delivering Cloud Native and Hybrid Cloud hosted applications.

There are two key capabilities of DevOps at its core: *continuous integration* and *continuous delivery*. Without these two capabilities, there is no DevOps, and they should be considered essential to DevOps adoption, with all others being extensions, or supporting capabilities. These two concepts focus on minimizing *cycle time*. Let's revisit the definition of *cycle time*.

NOTE Cycle time: The time from the inception of a requirement or user story to when that capability is in the hands of the customer, or at least is integrated, tested, and ready to be deployed to the customer.

Continuous Integration

Delivering a software application or system today involves multiple teams of developers working on separate components of the application. Typically, the completed application also needs to interact with other applications or services to perform its functions. Some of these external applications or services may be legacy applications that exist in the enterprise, or they may be external third-party services. There is, as a result, an inherent need for developers to integrate their work with components built by other development teams and with other applications and services.

This need makes integration an essential and complex task in the software development lifecycle. The process of doing this at a regular cadence is commonly referred to as *continuous integration*, and it is a key practice from Agile. In traditional development processes, integration was a secondary set of tasks conducted after the components (or sometimes the complete application) were built. This sequence was inherently costly and unpredictable, as the incompatibilities and defects that tend to be discovered only during integration were discovered late in the development process. The result was typically a significant increase in rework and risk.

The Agile movement introduced a logical step to help reduce this risk by integrating components continuously (or as continuously as possible). In this step,

[3] https://www.ibm.com/devops/method/

developers integrate their work with the rest of the development team regularly (at least daily) and test the integrated work. In the case of enterprise systems, which span multiple platforms, applications, or services, developers also integrate with other systems and services as often as possible. An example of Continuous Integration across multiple teams and components is shown in Figure 1-2.

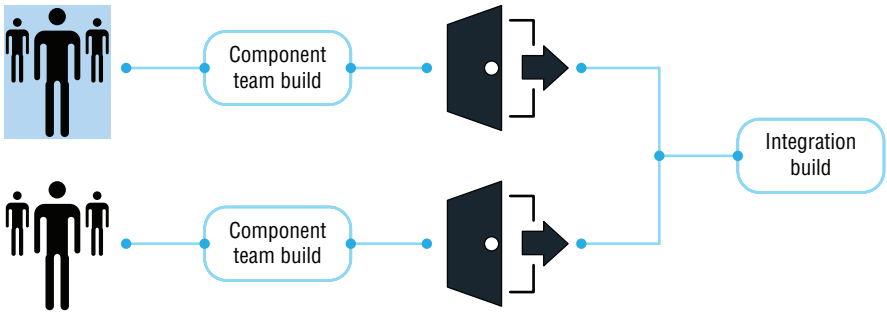

Figure 1-2: Continuous integration

These steps to integrate results can lead to early discovery and exposure of integration risks. In the case of enterprise systems, they can also expose known and unknown dependencies related to either technology or scheduling that may be at risk. As these practices have matured, some organizations have adopted continuous integration practices that developers follow every time they check in code. In the most mature organizations, continuous integration has led to capabilities for continuous delivery in which the code and components are not only integrated but are also delivered to a production-like environment for testing and verification. I'll discuss this in the next section.

The demands placed by business and customers on development organizations have driven the wide adoption by development teams of Agile development practices. These practices are aimed at reducing the gap between the business (or the customers) and the development teams. They work primarily in three ways:

- By breaking the development effort into small chunks of work that can be completed in time-bound iterations. This allows developers to identify and resolve risk earlier than when they undertake entire projects or larger portions of projects.
- By including contact with the end user or a surrogate representing the user into the development iterations. This helps give developers a better

understanding of the user's needs and allows for changing needs to be more quickly accommodated.

- By releasing software at the end of every iteration. This allows developers to demonstrate regularly what they have built in order to obtain user feedback.

As described, continuous integration is one of the tenets of such Agile development. It allows for developers to integrate their software components with components that are being developed by others—either internally or externally—on a regular basis, to allow for early identification of risks.

Practices of Continuous Integration

Martin Fowler, a signatory of what is known as the *Agile Manifesto*, is a thought leader in the development of continuous integration processes. He has broken down the concept into ten practices, which are described here.

1. **Maintain a single-source repository.** Whether managing code or any file, it is critical to use version management tools to manage the source base that allows multi-user access and streaming, or branching and merging, and that allows multiple developers in distributed locations to work on the same set of files. With any multi-platform development effort, using a common, cross-platform, single-source repository becomes even more important. If such a repository is not implemented across platforms, any platform left isolated (System z or Mobile, for example) will not be able to participate in continuous integration practices. Integration with any work conducted on the isolated platform will become an after-effort, waterfall-style integration.

   This transition to a modern source-code repository represents a significant change for legacy system development teams that may have been using the same capability for years. However, a single source code management (SCM) tool is critical to allow the management of all artifacts, help break down the silos, and remove a key bottleneck.
2. **Automate the build.** Automating the build is what makes continuous integration continuous. Additionally, it should be possible to coordinate the build across multiple platforms, when required.
3. **Make your build self-testing.** Just as builds need to be automated, so does the testing. The goal of continuous integration is not only to integrate the work of teams but also to see if the application or system being built is functioning and performing as expected. This requires

that a suite of automated test scripts be built for unit-test level and for the component and application level. In true continuous integration, developers should be able to start an integration build by kicking off the right test suite when they commit the code. This process requires that the build scripts include the capability to build the software if needed, provision the test server, provision the test environment, deploy the built software to the test server, set up the test data, and run the right test scripts.

The requirement to have the environments to do the build, deploy it, and do the automated testing at any time helps improve the quality of the final code. This requires availability of system resources, the willingness to run large numbers of automated tests on a regular basis, and the development of the automated tests.

4. **Ensure that everyone commits to the mainline every day.** The goal of having every developer, across all components and all development environments, commit their code to the mainline of their development streams every day is to help ensure that integrations remain as simple as possible. Even today, many developers work independently on their code changes until the final build, which is when they realize their work is impacted by the work of other developers. This can lead to delays in releasing functions or to last-minute changes that have not been properly tested being deployed into production. Regular integration of code can help ensure that these dependencies are identified sooner so the development team can handle them in a timely manner and without time constraints.
5. **Ensure that every commit builds the mainline on an integration machine.** This is a second part of Practice 4. Making sure that every commit is built and that automated regression tests are run can help ensure that problems are found and resolved earlier in the development cycle.
6. **Keep the build fast.** Virtually nothing impedes continuous integration more than a build that takes extremely long to run. Builds with modern tools are generally fast due to the standard practice of building only changed files.
7. **Test in a clone of the production environment.** Testing in an environment that does not accurately represent the production system leaves a lot of risk in the system. The goal of this practice, then, is to test in a clone of the production environment. It is not always possible, however, to create a clone of an entire multi-server environment just for testing.

It is even harder to create a clone environment with other workloads running on it.

Instead, this practice requires the creation of what is known as a *production-like* environment. In terms of specifications, this environment should be as close to the production environment as possible. It should also be subject to proper test data management. A test environment should not contain production data because in many cases that data needs to be masked. Proper test data management can also reduce the size and complexity of the test environment.

A complex system with multiple components—both pre-existing (such as other services and applications) and new components being developed—also creates challenges. All the components, services, and systems that applications need to access and interact with may not be available for running tests. This may occur for multiple reasons: the component, service, or system may not have been built yet; it may have been built but is available only as a production system that cannot be tested with non-production data; or it may have a cost associated with its use. For third-party services, for example, cost can become a major issue.

8. **Make it easy for anyone to get the latest executable.** Anyone associated with the project should have access to what is built and should be provided with a way to interact with it. This allows validation of what is being built against what was expected.
9. **Make sure everyone can see what is happening.** This is a communication-and-collaboration-related best practice, rather than one related to continuous integration. However, its importance to teams practicing continuous integration cannot be discounted. Visibility to the progress of continuous-integration builds via a central portal or dashboards can provide information to all practitioners.

   This can boost morale and help build the sense of working as a team with a common goal. If challenges occur, visibility can provide the impetus for people to step in and help other practitioners or teams. Visibility via a common team portal is especially important for teams that are not collocated—but it is also key for collocated teams and for cross-platform teams that work on different components of a project. This visibility should extend all the way back to the Business. As described in the earlier section on the *Business View*, visibility into the current status of the applications and services being delivered is a critical need of the business.

10. **Automate deployment.** Continuous integration naturally leads to the concept and practice of continuous delivery—the process of automating the deployment of software to test, system testing, staging, and production environments.

Continuous Delivery

Continuous delivery simply involves taking the concept of continuous integration to the next step. Once the application is built, at the end of every continuous integration build, it is delivered to the next stage in the application delivery lifecycle. It is delivered to the Quality Assurance (QA) team for testing and then to the operations team for delivery to the production system. The goal of continuous delivery is to get the new features that the developers are creating out to the customers and users as soon as possible. Now, all builds that come out of a continuous integration effort do not need to go to QA; only the "good" ones with functionality that is at a stage of development where it can be tested need to go to QA.

Similarly, all the builds that go through QA do not need to go to production. Only those that are ready to be delivered to the users, in terms of functionality, stability, and other non-functional requirements (NFRs) should be delivered to production. To test whether the builds coming out are production-ready, they should be delivered to a staging or test area that is production-like. This practice of regularly delivering the application being developed to QA and operations for validation and potential release to customers is referred to as *continuous delivery*.

Continuous delivery requires the creation of a delivery pipeline (as shown in Figure 1-3), with the core capability that automates the delivery pipeline being continuous delivery. As continuous integration produces builds at a steady pace, these builds need to be rapidly progressed to other environments in the delivery pipeline. Builds need to be deployed to the test environment to perform tests, to the integration environment for integration builds and integration testing, and so on, all the way to production. Continuous delivery facilitates deployment of applications from one environment to the next, as and when deployment is needed.

Continuous delivery, however, is not as simple as just moving files around. It requires orchestrating the deployments of code, content, applications, middleware and environment configurations, and process changes, as shown in Figure 1-4.

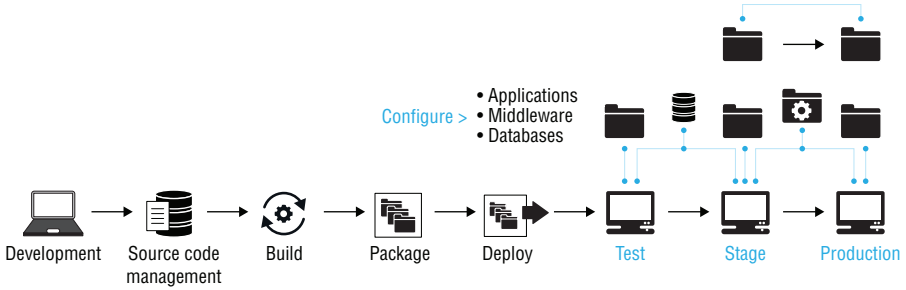

Figure 1-3: A delivery pipeline

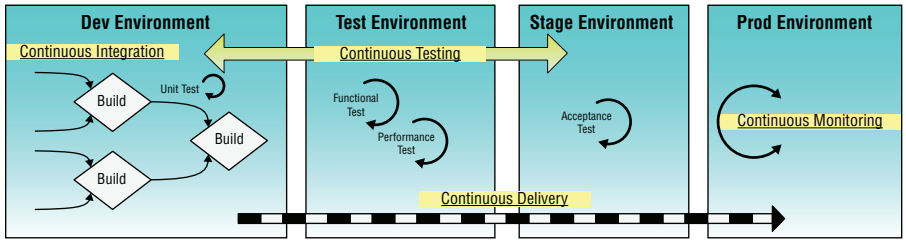

Figure 1-4: Continuous delivery

With regard to continuous delivery, there are two key points to remember:

- It does not mean deployment of every change out to production, a process commonly known as *continuous deployment*. Instead, continuous delivery is not a process but rather a capability to deploy to any environment, at any time, as needed. (I will discuss this more in the next section.)
- It does not always mean deploying a complete application. What is deployed may be the full application, one or many application components, application content, application or middleware configuration changes, or the environment to which the application is being deployed. It may also be any combination of these.

Two of the ten practices of continuous integration form the link to, and the necessity for, continuous delivery:

- Testing in a clone of the production environment
- Automating deployment

While testing in a clone of the production environment (the seventh practice) may be a testing practice, it also requires continuous delivery capabilities to deliver the new build to the clone test environment. This delivery may require provisioning the test environment and any virtualized instances of services and applications. It may also require the positioning of relevant test data, in addition to the actual deployment of the application to the right test environment.

The tenth practice of continuous integration, automating deployment, is the core practice of continuous delivery; it is not possible to achieve continuous delivery without automation of the deployment process. Whether the goal is to deploy the complete application or only one component or configuration change, continuous delivery requires having tools and processes in place to deploy, as and when needed, to any environment in the delivery pipeline.

Practicing continuous delivery also tests the actual deployment process. It is not unusual for organizations to suffer severe issues when deploying an application to production (as I discussed earlier). However, it is possible to uncover these issues early in the delivery lifecycle by automating the deployment process and validating it by deploying multiple times to production-like environments in pre-production.

Continuous Delivery versus Continuous Deployment

In the past, companies like Flickr posted on their blogs[4] how many *deploys* they had so far in a particular day or week. Looking at an organization that deploys to production 89 times in a week can be very intimidating. More importantly, it begs the question, "What do you deploy to production 89 times in a week?"

This is a scenario that may actually keep some people away from adopting DevOps practices, because they believe that they have to deploy every change to production. That is certainly not the case. First, you need to understand what is being deployed here, and second (and more importantly), you need to understand that this is not applicable, necessary, or even feasible for every organization.

What Do You Deploy 89 Times a Week? When organizations say they are doing double-digit deploys to production every day, it does not mean that they are delivering dozens of new features or bug fixes every day! What these companies have adopted is true and full-fledged continuous deployment. This means that every change by every developer works its way out

[4] http://code.flickr.net

to production. These may not be complete features; several such changes by multiple developers, over a matter of days, may make up a complete usable feature. They may not be visible to a customer at all; it is only after the complete feature is available and tested that it becomes visible. Then, too, it may be a part of an A-B test effort, so only a few customers will ever see it. The deployment may also be a simple configuration or database schema change that is never seen by anyone, but that changes some performance or behavior. Yet another scenario is where the deployment involves a new environment change and not an application change at all—an operating system (OS) or middleware patch, an OS- or middleware-level configuration change, a new database schema version, an entire new architectural topology of nodes, and so on.

Such a process is not viable for many organizations. Some organizations may have some (water-Scrum-fall like) requirements and policies that require a manual approval process before deployment to production. Others may require a *segregation of duties*, which mandates that the person to deploy to production is a different person or team from the one that contributes to the development of the deployable asset.

To Continuously Deploy or Not? There is still confusion among people between the concepts of continuous delivery and continuous deployment.

> *Continuous delivery doesn't mean every change is deployed to production ASAP. It means every change is proven to be deployable at any time.*
>
> —Carl Caum (Caum, 2013)

This tweet by Carl Caum, in a simple (less than 140-character) sentence, captures the essence of what *should* be done versus what *may* be done by an organization. Going by this distinction, continuous delivery is a *must*, while continuous deployment is an *option*. Having the capability to continuously deploy is more important than actually doing it in a continuous manner out to production (the key words here being *to production*). These terms are, unfortunately, still used interchangeably by most people.

What is required is the tested and validated capability to deploy to any environment in your delivery lifecycle—all the way out to production. You may only continuously deploy to an environment before *Prod* (lower environments)—for example, User Acceptance Testing (UAT), Pre-prod…, but the environments you deploy to should be *production-like*, so you know, with

very high confidence, that the final deploy to production will work without issues when you actually deploy to Prod.

What you continuously deliver should be every change to Dev and QA environments and other (lower) non-production environments. What you finally choose to deploy to Prod will typically be a full feature or set of features, or a full application or service.

Supporting Practices

Other than the two core practices of DevOps—continuous integration and continuous delivery (you are not doing DevOps without both being adopted)—there are several *supporting* practices. These have been developed to support and enable the two core practices. Following are some of these practices, which are considered to be supporting but essential.

Infrastructure as Code

MASTER OF THE OPS UNIVERSE

Imagine a seasoned operations engineer (neck beard and all). Over his career, he has most certainly developed a toolkit of scripts that he can use, with minor changes, to perform all his regular tasks of provisioning and managing the plethora of environments he has seen and dealt with. When it comes to configurations, he knows all the admin consoles he deals with like the back of his hand. He can log in and make the exact tweaks to application server configs that are needed to address the issues he is facing. For database-related issues, he knows exactly who to call and that the DBA has mastered his end of the deal as well as he has his. He has things down to a routine. He knows exactly when the next application release is due. He knows when to expect the next update to the OS. He is the master of his universe.

As systems have become virtualized and as developers have started practicing continuous integration (CI), things have started to change. The number of environments, and their instances that Ops engineers have to deal with, have increased by several orders of magnitude. Developers no longer release updates and new versions every few months; they are pumping out CI builds daily—in fact, multiple builds a day. All of these builds need to be tested and

validated. That requires new environment instances to be spun up, fast. These builds also often come with configuration changes. Logging into consoles to make each one of these changes individually is no longer a viable option. Furthermore, the need for speed is critical. Developers' builds are creating a backlog, as the environments to even test them on are not available as needed. *Houston, we have a problem.*

Let's start by revisiting two concepts:

1. **Cycle time.** *Cycle time* is defined as the average time taken from when a new requirement is approved, a change request is requested, or a bug that needs to be fixed via a patch is identified, to when it is delivered to production. Agile organizations want the delivery cycle time to be the bare minimum. This is what limits their ability to release new features and fixes to customers. Organizations like Etsy have cycle time down to minutes! While this is not possible for enterprise applications, the current cycle time of weeks or sometimes even months is absolutely unacceptable.
2. **Versioning environments.** The need to maintain multiple configurations and patch levels of environments that are now needed by development, on demand, requires Ops to modify how they handle change and maintain these environments. Any change Ops makes to an environment, whether it is applying a patch or making a configuration change, should be viewed as creating a new *version* of the environment, not just tweaking a config setting via a console. The only way this can be managed properly is by applying all changes via scripts. These scripts, when executed, would create a new version of the environment they are executed on. This process streamlines and simplifies change management, allowing it to scale, while keeping Ops best practices Information Technology Infrastructure Library (ITIL) and IT Service Management (ITSM) intact.

The solution to addressing both of these needs—minimizing cycle time and versioning environments—can be addressed by capturing and managing infrastructure as code. Spinning up a new virtual environment or a new version of the environment then becomes a matter of executing a script that can create and provision an image or set of images—all the way from the OS to the complete application stack being installed and configured. What took hours now takes minutes.

Versioning these scripts as you would version code in an SCM system allows for proper configuration management. Creating a new version of an environment now involves checking out the right scripts and making the necessary changes to the scripts—to patch the OS, change an app server setting, or install a new version of the application—and then checking the scripts back in as a new version of the environment, before executing it.

NOTE Without infrastructure as code, Ops can very easily become the "fall" in water-Scrum-fall.

Several automation frameworks have emerged to enable the capturing and management of infrastructure as code. The popular frameworks include Chef, Puppet, Salt, and Ansible.

With the evolution of the cloud, IT is now going to complete *software-defined environments* (*SDEs*). This takes the definition, versioning, and maintenance of complete environments as code. Technologies like OpenStack CloudFormation (for Amazon Web Services) are the leaders. OpenStack, for example, allows for *full stack* environments to be defined as software using Heat patterns, which can be versioned, provisioned, and configured using the likes of Chef and Salt, as needed. This also allows for the management of these environments at scale. No longer are Ops practitioners focused on managing individual servers that have long lifetimes; they are now managing large numbers of servers that are *transient* in their existence, and provisioned and de-provisioned on demand. This scale and agility can only be achieved with SDEs.

NOTE In a software-defined environment world, servers are "cattle," not "pets" (McCance, 2012) and (Bias, 2012).

Continuous Feedback

If you step back and look at *continuous feedback* in a holistic sense, it essentially means getting feedback from each functional area of the delivery pipeline to the areas to its left. So, developers provide feedback as they develop and deliver code, back to architects, analysts, and lines of business; testers provide feedback, through continuous testing to developers, architects, analysts and lines of business; and finally, Ops provides feedback to QA, testers, developers, architects, analysts, and lines of business, as well as everyone else who is a stakeholder.

The purpose of continuous feedback is to validate that the code produced and integrated with code from other developers and with other components of the application functions and performs as designed. Once the application has been deployed to a production system, it is also a goal to monitor that application to ensure that it functions and performs as designed in a production environment, as it is being used by end-users. This is essential to enable continuous improvement and quality. It is the core of Deming's PDCA cycle, as it provides the input to determine what to change and how to act.

NOTE Continuous integration and delivery are both (almost) meaningless without continuous feedback. Not having testing and monitoring in a continuous manner, and therefore not knowing how the application is performing in production, makes the entire process of DevOps moot. What good is having a streamlined continuous delivery process if the only way you find out that your application's functionality or performance are below par is via a ticket opened by a disgruntled user?

This brings me to the two practices of DevOps that are required to enable continuous feedback: continuous testing and continuous monitoring.

Continuous Testing *Continuous testing* is the capability for testing the application, the environment, and the delivery process at every stage of the delivery pipeline for the application being delivered. The items tested and the kinds of tests conducted can change depending on the stage of the delivery lifecycle. Continuous testing is really intertwined into the processes of continuous integration and continuous delivery, if done properly. Let's look at how this works in detail.

Individual developers work to create code. Fixing defects, adding new features, enhancing features, or making the code perform faster are some of the many tasks (work items) they may be working on. When done, they run unit tests on their own code and then deliver their code and integrate it with the work done by other developers on their team, as well as with unchanged code their team owns (*continuous integration*). Once the integration is done, they do unit tests on the integrated code. They may run other tests such as white box security tests, code performance tests, and so on. This work is then delivered to the common integration area of the team of teams—integrating the work of all the teams working on the project and all the code components that make up the service, application, or system being developed.

This is the essence of the process of continuous integration. What makes this process continuous is where an individual developer's code is integrated with that of their team, as and when they check in the code and it is delivered for integration. The important point to note here is the goal of the continuous integration process: to validate that the code integrates at all levels without error and that all tests run by developers run without error. Thus, continuous testing starts right with the developers.

After validating that the complete application (or service or system) is built without error, the application is delivered to the QA area. This delivery of code from the Dev or development environment to the QA environment is the first major step in continuous delivery. There is continuous delivery happening as the developers deliver their code to their team's integration space and to the project's integration space, but this is limited to being within the Dev space. There is no new environment to target.

When delivering to QA, I am speaking of a complete transition from one environment to another. QA has its own environment on which to run its suites of functional and performance tests. DevOps principles demand that this environment be production-like. In addition, QA may also need new data sets for each run of the suites of tests it runs. This may be one or more times every day as continuous integration leads to continuous delivery at a steady stream. This means that the continuous delivery process not only requires the processes to transition the code from Dev to QA, but also to refresh or provision new instances of QA's production-like environments, complete with the right configurations and associated test data to run the tests against. This makes continuous delivery a more complex process than just copying code over. The key point to note is that the goal of continuous delivery is to get the code ready for test, and for release, and to get the application to the right environment—continuously, so that it can be tested continuously.

If you extend the process described here to delivering the service, application, or system to a staging and eventually a production environment, the process and goal remain the same. The Ops team wants to run their own set of smoke tests, acceptance tests, and system stability tests before they deliver the application to the *must-stay-up-at-all-costs* production environment. That is done using a staging environment. This is a production-like environment that needs to be provisioned just like the QA environment. It needs to have the necessary scripts and test data for acceptance and performance tests that Ops will run. Only when this last phase of continuous testing is complete is the application delivered to production. Continuous delivery processes, hence, also perform the tasks of providing staging and production environments and delivering the application.

To delve more into this process, continuous testing is achieved by testing all aspects of the application and environment, including, but not limited to, the following:

- Unit testing
- Functional testing
- Performance testing
- Integration testing
- System integration testing
- Security testing
- User acceptance testing

In continuous testing, the biggest challenge is that some of the applications, services, and data sources that are required to perform some tests may not be available. Alternatively, even if they are available, the cost associated with using them may prohibit running tests on an ongoing basis. Furthermore, the costs of maintaining large test environments to serve all teams developing multiple applications in parallel can also be high.

The solution is to introduce the practice known as *test virtualization* (see Figure 1-5). This practice replaces actual applications, services, and data sources that the application must communicate and interact with during the test, with virtual *stubs*. These virtual instances make it possible to test applications for functionality, integration, and performance without making the entire ecosystem available. This virtualization can be utilized to perform the myriad types of testing listed earlier.

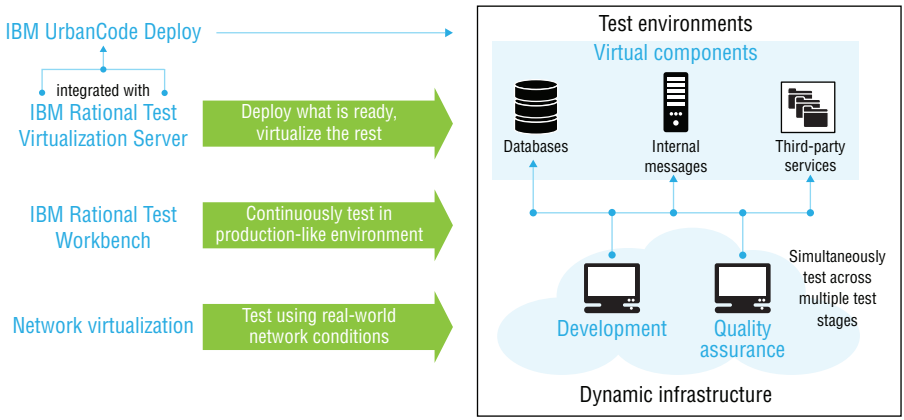

Figure 1-5: Example of test virtualization

When it comes to testing in the context of DevOps, in addition to continuous testing, there is also the practice of shift left testing, which I will examine in the "Shift Left" section, later in this chapter.

Continuous Monitoring In production, the Ops team manages and ensures that an application is performing as desired and the environment is stable via continuous monitoring. Ops teams have their own tools to monitor their environments and running systems. Ultimately, the Ops team needs to ensure that the applications are performing, from the process level down to levels that are lower than what system-monitoring tools would allow. This requires that Ops teams use tools that can monitor application performance and issues. It may also require that they work with Dev to incorporate self-monitoring or analytics-gathering capabilities right into the applications that are being built. This would allow for true end-to-end monitoring, continuously.

As the technology in this space has grown, there has also been the emergence of tools and services that monitor application behavior and user sentiment, providing even finer-grained feedback that is useful to developers and the line of business.

In a nutshell, continuous monitoring requires the capture and analysis of metrics in four areas:

- Application performance
- System performance
- Application user behavior
- User sentiment

It is, however, essential that the Ops teams not just gather this data but also run analytics on it. Furthermore, they must make their feedback consumable by their target audience, from deep technical Ops practitioners, like performance engineers, to non-technical line-of-business stakeholders. Data is of no value unless it is consumable. Good data, and even better, good analytics on the data, can truly enable continuous improvement, as decisions at all levels of the delivery pipeline—from line of business, to developers, to testers—can now be data driven.

THE FUTURE OF FEEDBACK IS COGNITIVE

With the advent of cognitive capabilities like IBM Watson, tremendous capabilities are being brought to market in the area of predictive analytics

of this feedback data. Data from user behavior, application behavior, and system behavior can now be analyzed, leveraging cognitive techniques to deliver predictive results, from predictive failure of systems, to predictive behavior of (happy or disgruntled) customers. Predictive analysis can result in businesses acting preemptively to prevent outages and disgruntlement.

Continuous Business Planning

The DevOps practice of *continuous business planning* focuses on the lines of business and their planning processes. Businesses need to be agile and able to react quickly to customer feedback. To achieve this, many businesses today employ Lean thinking techniques. These techniques involve starting small by identifying the outcomes and resources needed to test the business vision or value and then continuously adapting and adjusting based on customer feedback.

To achieve these goals, organizations measure the current baseline state, find out what customers really want, and then shift direction by updating their business plans accordingly, allowing them to make continuous trade-off decisions in a resource-constrained environment.

There has been a lot of work done in this space to leverage techniques made popular by the Lean startup movement, and described by Eric Reis in his book, *The Lean Startup*. The set of techniques, like delivering a minimum viable product, that are introduced by Reis in his book are becoming popular with businesses wanting to experiment with new markets and new business models, without having to make complete plans for delivering complex IT systems for these new areas. I will discuss this in more detail in Chapter 4.

The latest addition to the arsenal of capabilities available to ensure that you are not just building the deliverable right, but also building the right deliverables, is design thinking. Like Lean and Agile, design thinking has been used in industrial design for physical products for decades, in various levels of its evolution. It became mainstream with Peter Rowe's 1987 book, aptly named *Design Thinking*. What is new is its adaptation to IT and, specifically, application design, with a focus on user experience. Design thinking will be explored in more detail in Chapter 4.

Collaborative Development

Collaborative development was made popular by IBM, primarily as a practice supported by its *Collaborative Lifecycle Management (CLM)* tool suite. The

practice is essentially in place to ensure that organizations with large, distributed teams enable visibility between, and collaboration among, cross-function practitioners and teams of teams, across silos. This is achieved by ensuring two capabilities across the delivery pipeline:

- Provision of access and visibility by practitioners not just to artifacts, work items, and metrics related to their functional area, but across all functional areas into which they need to have visibility (of course, access is managed by role and security needs).
- Seamless handoff of artifacts from one practitioner or team to another. This should be possible across functional boundaries, and should not require any translation or transformation of the artifact, in order for it to be consumed.

These capabilities can only be achieved by having a set of integrated tools utilized by practitioners and teams, across the delivery pipeline.

If you look at DevOps as a cultural movement, where the fostering of communication, collaboration, and trust are the core tenets you are striving for, then collaborative development may be seen as a core capability of DevOps. There is no better way to promote communication, collaboration, and trust than by enabling practitioners to communicate with other practitioners using a common tool (which is not email).

This can be achieved using tools such as Slack or Rational Team Concert, which are becoming popular. The collaboration can be further enhanced by leveraging in-tool collaboration around work items, enabling practitioners to move work items between each other, add notes, attach code change sets, and have visibility into what other team members have worked on, or are currently working on, that impacts their own work.

Speaking of visibility, nothing fosters trust more than full visibility. If a tester has visibility into what a developer is unit testing, the developer knows that she cannot commit code without proper unit testing.

NOTE Total visibility drives total trust.

At our company, we will no longer require filing expense claims. You can spend whatever you want and we will reimburse you. No questions asked. All we ask you to do is to post your receipt on an open Wiki page which every employee in the company can see. Trust me, you will spend wisely.

—CEO of a Silicon Valley startup

Shift Left

Shift left as a concept also has its origins in Lean. The basic idea here is to improve quality by moving tasks that can impact quality to as early in the lifecycle as possible. This is done across the lifecycle. The underlying premise is that the earlier quality issues are caught, the earlier their root cause can be identified and addressed.

NOTE There is a well-known axiom in the QA space that if it takes one cent to catch and fix a defect or problem in the requirements stage, it will cost ten cents to fix the same in development, one dollar to fix in testing, and ten dollars to fix in production (Rice, 2009).

These are, of course, illustrative numbers and are not based on some statistical analysis of actual costs; however, the logic is sound. Shifting left the tasks that can identify defects and problems early saves money and improves quality.

From a DevOps culture perspective, you can also look at shift left as an approach used to improve collaboration and communication by engaging practitioners from functions that are to the right in the delivery pipeline, earlier in the lifecycle.

DEVOPS OR COUPLES' COUNSELING?

I had been asked by the architect on the account to meet with the Director of Dev and the Director of Ops for a client of his. We met for lunch, with the architect and me on one side of the table and the two directors on the other. I knew right away that all was not well on their home front. They were leaning away from each other. The Dev director complained about how Ops was not agile, and the Ops director said that Dev sent them garbage that would not even run without crashing servers. They even looked at their hands while speaking about the other. I felt I was in couples' counseling.

The solution plan I recommended to them was to begin with small steps, by shifting left when Ops was engaged. Their main challenge was a total lack of visibility between the Dev and Ops teams, till it was time to deploy to production. The suggestion I made was to pick one critical project and,

continued

continued

> once a week, have the Ops team send one resource to the Dev team's daily standup meeting and have them just listen, without needing to engage, and see if things improved. I had a follow-up meeting with the same two directors less than three months later at a conference. They were happy to report that the Ops team now had a presence at the daily standup meeting, and Ops not only listened, but actively participated, sharing their progress, plans, and blockers. Ops engagement had shifted left. They had achieved collaboration.

For maximum impact on quality improvement, there are two major areas where shift left needs to be adopted in the delivery pipeline.

Shift Left Testing

Engaging testers early, right from the requirements stage, better prepares them for what they will need to test, and in turn, they can also ensure that the requirements being written are testable. The goal, however, is to start testing earlier in the lifecycle. The practice of shift left testing, as it is gaining traction in the industry, is focused above all on ensuring integration testing earlier in the lifecycle. While other forms of testing (as described in the section, "Continuous Testing") are important to shift to earlier in the lifecycle, the value of shifting integration testing earlier is the highest.

As teams practice continuous integration, testing those integration points to identify integration and architectural deficiencies early has a significant impact on quality. What is the use of having perfectly functioning and performing services or components, if they don't work with other services and components when integrated? In order to achieve integration testing early in the lifecycle, test virtualization becomes a prerequisite, as all the services or components required in order to complete testing may not be available when needed. Test virtualization enables the stubbing out of these unavailable services with virtual instances, enabling integration—and other—testing early in the lifecycle, thus achieving shift left testing. You need to shift left to achieve the proverbial "Test Early, Test Often" goal.

Shift Left Operations Concerns

As described in the anecdote at the beginning of this section, the Ops team is usually seen as a separate silo in the delivery lifecycle. They are typically engaged at the beginning of projects, as operational requirements are determined, and then left disconnected from the Dev efforts, till it comes time to start operational

readiness, before handoff to production. Engaging Ops early in the lifecycle and having them participate in the Dev-test cycle prevents challenges that manifest during deployment to production, if Ops is engaged late. Engaging Ops early makes them aware of what is being delivered and how it will result in changes to Ops environments, as the needs may have deviated from the *as designed* state.

Engaging Ops early also helps to ensure that the production-like environments Dev and test are deploying to during Dev-test, are truly still production-like and have not drifted away from real production environments. Lastly, engaging Ops early also ensures that the deployment processes and procedure being developed by Dev teams are consumable by Ops. In the pre-DevOps days, one of the biggest challenges with deployment to production on a release weekend was the fact that deployment processes had never been used or tested by Ops before. Ensuring that these processes are tested over and over again as code is deployed to non-Prod environments—early and often, using the same processes and procedures that Ops will use—ensures that they will work in production.

A significant impact of shifting left is the change that happens in the roles of the practitioners. These changes happen subtly and over time, resulting in unintended consequences when it comes to skills needed and, eventually, headcount distribution across the delivery pipeline.

As responsibilities shift left, the role of the practitioner changes from that of a *doer* to that of a *service provider.* Testers may no longer be the ones doing the tests; instead, they become providers for test automation, which can be self-served by the developers. Similarly, for Ops practitioners, they are no longer the ones running around building, provisioning, and de-provisioning servers. Instead, they build server images, manage server instances, and respond to issues. Dev, test, and other practitioners provision, configure, and de-provision instances of servers, on demand, leveraging the self-service access provided and managed by Ops teams. This raises the abstraction at which the testers and Ops now work and perform. Consequently, it impacts the skills they need, and the numbers of resources that may be needed.

Architecture and Risk Mitigation

ARCHITECTURAL THINKING

When I joined Rational Software in the mid-'90s, the focus on architecture was imbibed into my thinking. With the methodology "Three Amigos" Grady Booch, James Rumbaugh, and Ivar Jacobson developing UML

continued

continued

> (Jim joined Rational Software just before I did. We still had Booch's Clouds for Objects) and Philippe Kruchten developing his 4+1 View Model of Software Architecture (Kruchten, 2002), architectural thinking was, and is, in my bloodstream.

The area of application delivery that is finally beginning to get the attention it needs, in order to get the full promise of DevOps realized, is architecture. You cannot achieve continuous delivery with large, monolithic systems. While architectural refactoring was largely ignored in the early days of DevOps, it is going mainstream now, mainly thanks to the evolution of microservices (or what are referred to as *12-factor apps*).[5]

While the debate is still ongoing over whether microservices can truly deliver the value for every kind of application, the attention that microservices have received has revived a much-needed focus on architecture. If you truly understand 12-factor apps, their focus on web apps and Software as a Service is self-evident. They may not add value to apps and systems that are large, complex, data-heavy legacy systems, without expensive refactoring of code and data. That investment is viable and necessary only if those systems are being modernized into cloud-native apps. Microservices and 12-factor apps will be discussed in more depth in Chapter 5.

The architectural transformation needed to achieve continuous delivery, irrespective of whether microservices are used, is to enable the delivery of changes in small batches—thus, reducing batch size. A *batch* is the number of changes being delivered in each cycle or sprint. These changes include any and all changes—code, configurations, infrastructure, data, data-schemas, scripts, deployment processes, and so on—that encompass a full Dev-to-Ops cycle. (Remember, not all changes are deployed to production every time.) Reducing batch size is imperative to do the following:

- Reduce risk
- Improve quality
- Enable faster delivery

These benefits are self-evident. The most effective way to manage risk and quality, while increasing speed, is to reduce the batch size in each iteration or *sprint*. This is a mind shift to deliver smaller, more frequent new versions.

[5] `http://12factor.net`

As you reduce batch size, there is less to test and validate in each cycle; there is less to deploy; and, because there is less change, there is lower risk. If challenges or issues are identified, their impact is also limited by the smaller batch size, making mitigation easier, via fixes or rollbacks.

Continuous Improvement

At the end of the day, the heart of DevOps lies in achieving continuous improvement. No matter where you start, at whatever maturity level, adopting DevOps is not a one-time project you undertake; it is an ongoing effort. The goal is to ultimately become a learning organization, as envisioned by Peter Senge in the '90s (David A. GarvinAmy C. Edmondson, 2008). In the DevOps context, a *learning organization* is constantly learning from what it just delivered, and continuously improving. What do you improve? There are three areas of improvement:

- **The application.** Are the application changes that you just delivered functioning and performing as desired? What can you learn from the continuous feedback coming in to improve the app in the next iteration?
- **The environment.** Are the environments the application is running on performing and behaving as desired? Are the service level agreements (SLAs) being met? What can you learn from the continuous feedback that is coming in to improve the environments in the next iteration?
- **The process.** What can you learn from the experiences of the practitioners and stakeholders to improve the delivery processes themselves in the next iteration?

While most organizations have efforts ongoing to continuously improve the application being delivered, fewer organizations have the same level of rigor for continuously improving the operational environments, based on real metrics. Far fewer organizations have programs in place to continuously improve delivery processes. This is the case despite movements like Lean, and their incarnations like Agile's Scrum and the broader Lean startup, which have built into them what is needed to become a learning organization or team, and to be constantly improving at a process level.

Metrics

> *If you can't measure it, you can't manage it.*
>
> —Attributed to Peter Drucker

Irrespective of whether Peter Drucker actually said this, or of whether it is even accurate (Kaz, 2013), the fact remains that in order to manage and consequently improve something, you need to be able to measure some critical metrics: Key Performance Indicators (KPIs). You will need both a baseline measurement of these KPIs, marking the starting point, and ongoing measurements to see if improvement is indeed occurring. Not only do you need to measure that the needle is moving—and in a positive direction—but you also need to be able to understand cause and effect: which actions result in improving KPIs. If you are making several changes to people and processes, knowing which changes are actually resulting in improvement is critical.

Business Drivers

To know which metrics to measure and improve, you have to know the business drivers. What business impact are you striving for? Change, and even improvement for the sake of improvement, does not make good business sense. If you are going to invest in transforming an organization by adopting DevOps, knowing what business drivers need to be addressed is a prerequisite. It helps to determine which metrics matter, and thus, which capabilities to focus on and invest in. Focusing on speed alone means that you are taking a very myopic view of the world.

> *As a medical device manufacturer, quality always trumps speed for us. We would rather be late in releasing a device, than ever have to issue a recall. As you can imagine, recalling installed pacemakers is not a good situation for anyone.*
>
> —Director of QA at a medical device manufacturer

What KPIs or metrics should you measure and strive to improve? As I mentioned earlier, it all depends on business drivers. What are the lines of business asking you, the IT organization, to improve? (This may vary by line of business, even within the same organization.) Is it speed, quality, agility, ability to innovate, or cost reduction? Is it something at an even higher level, such as the ability to deploy new business models or capture new markets? Is it something at a lower level, such as reducing the mean time between failures (MTBF), or improving mean time to resolve (MTTR); or is it just lowering bug density in the code? Is it being able to develop a partner ecosystem with APIs? Is it reducing the time it takes to get all the approvals IT needs to

deliver a new app? Is it being able to attract more tech talent by participating in open source projects? (Everyone knows that it is the cool companies that contribute to open source projects.)

Here is a subset of core DevOps metrics that a division at IBM used to measure the impact of DevOps adoption. These metrics, shown in the following list, were all determined by the business drivers that this group needed to have an impact on (speed to market, market share, and improving profitability of the products they delivered).

- Project initiation
- Groomed backlog
- Overall time to development
- Composite build time
- Build Verification Test (BVT) availability
- Sprint test time
- Total deployment time
- Overall time to production
- Time between releases
- Time spent—innovation/maintenance (percentage)

DevOps: Culture

"Everyone is responsible for delivery to production." That is what the T-shirt says. I am giving it to everyone who is even remotely connected to my project. Of course, the analysts, designers, developers, testers, ops folk assigned to the project get it. But so do the people on the enterprise architecture team, the application architecture team, and the security guys. The people in the PMO definitely get one. I gave one to the janitor who has our floor—if the restroom is busted and an engineer wastes 20 minutes to use one on another floor, the janitor is now responsible for a delay in deployment to production. I gave one to the coffee machine maintenance guy. If the coffee machine is out of pods and we send one of the interns over to Starbucks, the coffee machine maintenance guy is now responsible for a delay. I FedEx'ed one to our CFO. If she can't manage the budget and furloughs even one of my contractors this December, like she did last year, she is now delaying deployment to production. The CIO gets it to keep my team out of email-jail. The CTO gets it for not delaying technology approvals. Heck, if my wife had not convinced me that it was a bad idea, I would have handed it to every

"significant other" who showed up at the company picnic. That, my friend, is what a DevOps culture means to me.

—VP at a large insurance company, defining DevOps culture

As I mentioned before, DevOps, at its heart, is a cultural movement. So, how do you change culture? Ultimately, even after all the process improvement and automation that can be introduced in an organization, the organization can only succeed at adopting the culture of DevOps if it is able to overcome the inherent cultural inertia. Organizations have inertia—an inherent resistance to change. Change is not easy, especially in large organizations where the cultural may have had years to develop and permeates across hundreds, if not thousands, of practitioners. These practitioners, as individuals, may appreciate the value of adopting DevOps, but as a collective, they resist change and thus have inertia. Overcoming this inertia is key. Cultural inertia can be exhibited by the following statements:

"This is the way we do things here."
"Yes, but changing X is not in my control."
"Nothing is broken in our processes. Why should we change?"
"You will need to talk to Y about that; WE cannot change how THEY work."
"Management will never allow that."
"Don't you know we are in a regulated industry?"
"DevOps is the new flavor of the month. Let's see how long this effort lasts."

Over time, organizations develop behaviors; teams and groups divide up actions and responsibilities along organizational lines; checks and balances are established in the name of governance but are not related to true governance at all; processes exist, but no one knows why—they are *just there*; reports are produced that no one reads anymore, but no one is willing to do away with them; bad things happened in the past and resulted in approval requirements to ensure they never happen again; and so on. All of these behaviors build up inertia in an organization's culture.

What kind of culture does DevOps adoption need? One of trust, communication, and collaboration. Adopting DevOps practices alone will not foster such a culture, nor will the practices take root and become ingrained in an organization's DNA unless such a culture begins to develop. It is a

chicken-and-egg situation that requires a concerted effort to overcome the cultural inertia. This cultural inertia can be overcome by addressing three areas:

1. **Visibility.** I discussed this at length earlier in this chapter, and its value cannot be ignored. There is no greater cause of mistrust than not having visibility into teams or practitioners that you have to engage with, and you are not sure what they did with the artifacts they are handing off to you.
2. **Effective communication.** Email and voicemail need to be done away with as sources of communication in a DevOps environment; so do project plan and status documents, slide decks, and spreadsheets. Communication needs to be live and peer-to-peer, not via email or tickets, or done through management. One practitioner should be able to communicate with any other practitioner she needs to, without having to go through a chain of command. These live communications should replace email, status updates, and collaboration, and they should be streaming. Tools like Slack, HipChat, Yammer, and Wrike are becoming very popular as a result.
3. **Common measurements.** Out of all that I've mentioned, the area that causes the most inertia is a lack of right measurements for practitioners and teams. People will not change their behaviors, unless the way they are being measured matches the new, desired behaviors. Furthermore, to deliver true collaboration and a sense of a single team working toward a singular set of goals across silos, these measurements of success should be the same among all practitioners. Dev, test, and Ops need to have common or at least similar metrics that their success is measured on. Everyone—and I mean everyone—has to be made responsible for deploying to production.

Summary

DevOps is now mainstream. While that is a given, not everyone has come to the same understanding of what DevOps is and, more importantly, how it should be adopted. The right answer is, unfortunately, "It depends." And it does. It depends on the business goals you are striving for; it depends on what the current maturity of practices is; and it depends on the rate of change your

organization is able to absorb. Change has to be adopted to achieve increased business value, but not at its expense. Any disruption results in dips in productivity, and that is also true for DevOps adoption.

Adopting DevOps is a journey that has to begin with the first step of identifying point A (your current state) and point B (your business goals). Once you have identified these points, you can develop an adoption roadmap to adopt the right practices and capabilities (the right plays) described in this chapter. How do you go about creating such an adoption roadmap? That is the topic of the next chapter.